Q 2]

a] Pseudo-code for selection sort

```
SELECTION-SORT (A)

for i = 1 to A.length - 1
    boundary = i
    for j = i + 1 to A.length
        if A[j] < A[boundary] and j != boundary
            boundary = j
    swap A[i] and A[boundary]
```

b] Loop Invariant
At the start of each iteration of the outer for loop, the subarray A[1..i-1] consists of i-1 elements which are smaller than all the elements in the remaining subarray A[j... n], sorted in ascending order.

c] Why does it run only for first n-1 elements?

→ i) From the loop invariant statement, we can conclude that for ~~the~~ every i^{th} iteration of the algorithm, all the elements to the left of the i^{th} index will be smaller than the elements to the right and in ~~the~~ a sorted order

~~ii) Thus, after the last iteration, the last element in the initial array~~

ii) Thus, during the last iteration, which is $i-1^{th}$, the last element will be sorted and be put in the right place and the list/array gets sorted.

iii) Since the final state is reached already on the ~~i^{th}~~ $n-1^{th}$ iteration, there is no need to continue further.

d] Best and Worst case of Selection Sort

→ i) Selection sort will always take one element at a time and compare it with all the elements in the already sorted array to the left in every iteration.

ii) The algorithm compares each element with all the other elements on its left, i.e., each nth element is compared with n-1 elements. This is because the algorithm does not have an exit condition for the inner for loop once the nth element's new index in the sorted subarray is found.

iii) Hence, the selection sort runs for the same number of times irrespective of the case. And from (ii), the best and worst case running time both will be $\Theta(n^2)$

~~Q4~~ Q 3

~~Let~~

After dividing the original array into three parts instead of two, we need to take into account three different combinations of possible maximum subarrays during the merge step of the algorithm.

~~Step~~ Assume that A1, A2 and A3 are three equal subarrays of A for this problem

1] Case 1:

When ~~maxi~~mum subarray belongs in A1 and A2

→ Here, we check the elements in both A1 and A2 while merging from start to end.

$$\text{cost} = \theta\left(\frac{2n}{3}\right) + \theta(1)$$

Assuming A1, A2 and A3 are uniformly split

2) Case 2

When maximum subarray is in A2 and A3.
Similar to case 1, we scan both arrays completely giving us the cost as $\theta(\frac{2n}{3}) + \theta(1)$

3) Case 3

When maximum subarray is in all three subarrays.
In this case we will end up navigating through all the elements.
$\therefore$ Cost $= \theta(n) + \theta(1)$

Note : We consider only these 3 cases because if the max. subarray is only in any one of the partitions, it will not have any effect on the time complexity since the partitions do not change anything and it remains a generic divide and conquer problem.
Thus, we ignore the other cases.

Now, from case ①, ② and ③,

$$T(n) = 3T(\frac{n}{3}) + \theta(\frac{2n}{3}) + \theta(1)$$
$$+ \theta(\frac{2n}{3}) + \theta(1) + \theta(n) + \theta(1)$$
$$= 3T(\frac{n}{3}) + \theta(n)$$

Thus, applying master's theory we get

$$\boxed{T(n) = \theta(n\log n)}$$

Q 5] Writing a recursive algorithm for finding min and max from array,
consider 3 parts.

1. If length of ~~sto~~ array is 1
2. Comparison between 2 elements at the lowest level / deepest level & evaluating min and max
3. Recursively breaking the array into halves and calling the min-max function

```
MinMax (A, start, end, min, max)
if start == end
        max = min = A[start]
else if start == end - 1
        if A[start] < A[end]
            if min > A[start]
                min = A[start]
            if max < A[end]
                max = A[end]
        else
            if min > A[end]
                min = A[end]
            if max < A[start]
                max = A[start]
        return
midpoint = (start + end) / 2
MinMax (A, start, midpoint, min, max)
MinMax (A, midpoint + 1, end, min, max)
return
```

Finding complexity,

Let $T(n)$ be the number of comparisons made by MinMax algo.

∴ The relation →

$$T(n) = T(n/2) + T(n/2) + 2 \quad (n > 2)$$
$$= 1 \quad (\text{when } n = 2)$$
$$= 1 \quad (\text{when } n = 1)$$

$$\therefore T(n) = 2T\left(\frac{n}{2}\right) + 2$$

When n is a power of 2,
$n = 2^k$ where, k is the height of the recursion tree

Solving the equation

$$T(n) = 2T\left(\frac{n}{2}\right) + 2$$
$$= 2\left[2T\left(\frac{n}{4}\right) + 2\right] + 2$$
$$= 4T\left(\frac{n}{4}\right) + 2 + 2(2)$$
$$= 4\left[2T\left(\frac{n}{8}\right) + 2\right] + 6$$
$$= 8T\left(\frac{n}{8}\right) + 14$$

~~Thus, when~~ $n =$
For each ~~pair~~ element in array we do 2 comparisons [Array size > 2]

∴ We have a total of $n-2$ comparisons for n elements. [while merging]

In addition, we have $\frac{n}{2}$ comparison at the deepest level of the tree.

$$\therefore \boxed{\text{Total comparisons} = (n-2) + \frac{n}{2} = \frac{3n}{2} - 2}$$